Part 5: Running a Mini-CA

In this part, we will learn how to use OpenSSL command-line utilities to run a mini-**Certificate Authority** (**CA**) that can issue certificates for internal usage in an organization. A mini-CA can be useful in organizations for establishing internal Public Key Infrastructure, gaining control of internally used certificates, and saving costs on ordering certificates from commercial CAs. The usage of a mini-CA will be illustrated using configuration files and command-line examples.

This part contains the following chapter:

- *Chapter 12, Running a Mini-CA*

12
Running a Mini-CA

In *Chapter 8*, *X.509 Certificates and PKI*, we learned about **Public Key Infrastructure** (**PKI**) based on X.509 certificates. In this chapter, we will learn how to use the `openssl ca` subcommand to run a mini-**Certificate Authority** (**CA**) that can issue certificates for internal usage in an organization. A mini-CA can be useful in organizations for establishing internal PKI, gaining control of internally used certificates, and saving costs on ordering certificates from commercial CAs. The usage of a mini-CA will be illustrated by command-line examples.

We are going to cover the following topics in this chapter:

- Understanding the `openssl ca` subcommand
- Generating a root CA certificate
- Generating an intermediate CA certificate
- Generating a certificate for a web server
- Generating a certificate for a web and email client
- Revoking certificates and generating CRLs
- Providing certificate revocation status via OCSP

Technical requirements

This chapter will contain commands that you can run on a command line. You will need the `openssl` command-line tool and the OpenSSL dynamic libraries.

We will use many configuration files and commands in this chapter. Those configuration files and commands, saved as Shell scripts, can be found here: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter12`.

Understanding the openssl ca subcommand

The `openssl ca` subcommand can be useful for running a mini-CA inside an organization. This kind of CA can, for instance, issue certificates for internal servers. Using an internal CA saves costs compared to using an external commercial CA. But it is not the only advantage. Many internal servers should not be exposed to access from the internet. This limitation hinders automatic server checks from the external CAs, which are needed to issue cheap or free certificates. Also, in some cases, it is undesirable to expose knowledge about the existence or name of the internal servers. When ordering a certificate from an external CA, you have to expose the internal server's name to the CA. Furthermore, the CA may publish the certificate information to a **Certificate Transparency** (**CT**) log, leading to even more unwanted information exposure about the company's internal servers.

Another reason to have an internal CA is to issue client certificates. Using a certificate issued by an internal CA on a public internet web server is not very convenient because every user of that web server has to install the internal CA certificate to their browser's trusted store. But a client certificate issued by an internal CA does not have this disadvantage because such a certificate is usually only used to authenticate against a few servers.

In some organizations, using an internal CA for internal purposes can give some degree of freedom, compared to using an external CA. For example, you may need approval from a security team to get an external certificate or from an accounting manager to get a paid certificate. With an internal CA, you can freely issue certificates for your internal needs without any approvals.

The `openssl ca` subcommand can take parameters for its work from the command line and the OpenSSL config file. The OpenSSL config file can store various options for the OpenSSL library and utilities. The default OpenSSL config file is supplied in the OpenSSL distribution as the `apps/openssl.cnf` file. During OpenSSL installation, the default config file will be installed into the `ssl` directory – for instance, as `/opt/openssl-3.0.0/ssl/openssl.cnf` or `/etc/ssl/openssl.cnf`. For `openssl ca` operations, it is usual to create a custom config file representing a particular CA, such as a root CA or an intermediate CA, instead of using the default OpenSSL config file.

It is preferable to specify most parameters for the `openssl ca` subcommand in the config file in order to avoid repeating them on the command line when issuing different certificates. Besides, some parameters can only be specified in the config file. If the same parameters are provided in both the config file and on the command line, the command line parameters take precedence.

Documentation about `openssl ca` parameters and config files can be found on the following `man` pages:

```
$ man openssl-ca
$ man 5ssl config
$ man x509v3_config
```

Along with the config file, `openssl ca` also uses other mandatory or optional files, such as certificate index files, serial number files, random seed files, and CRL number files.

The `openssl ca` subcommand can issue certificates, revoke certificates, and generate **Certificate Revocation Lists** (**CRLs**). When issuing or revoking a certificate, `openssl ca` updates the certificate index file, also known as the certificate database.

`openssl ca` can issue certificates with sequential or random serial numbers. If sequential serial numbers are used, the next serial number is stored and automatically updated in the serial number file. However, for security reasons, it is better to issue certificates with random serial numbers.

A CRL can be generated with or without a **CRL number**. A CRL number is basically a CRL version. It is a positive integer number that is incremented by one each time a CA issues its next full CRL. It is better to include a CRL number in a CRL because it helps to check CRL issuance continuity and makes it possible to generate **delta CRLs**. A delta CRL is a CRL that only contains changes since a particular release of an ordinary full CRL. The full CRL that the delta CRL is based on is called a **base CRL**. However, delta CRLs are not supported by `openssl ca` yet – you will need to use another tool if you want to generate them. If you generate a CRL with a CRL number, that number will be stored in the CRL number file and updated automatically by `openssl ca`.

Let's learn how to use the `openssl ca` subcommand. We will start by generating a root CA certificate.

Generating a root CA certificate

We will create some needed directories and files and then proceed with generating the root CA certificate:

1. First, we will make the `mini-ca` directory where we will put all our CA-related files:

```
$ mkdir mini-ca
$ cd mini-ca
```

2. Inside the `mini-ca` directory, we will create the `root` directory for the files related to the root CA:

```
$ mkdir root
$ cd root
```

3. Inside the `root` directory, we will create a directory for issued certificates, a certificate index file, and a CRL number file:

```
$ mkdir issued
$ echo -n >index.txt
$ echo 01 >crlnumber.txt
```

4. Now, we have to make a config file for our root CA; we will name this file `root.cnf`. We will start the config file with some mandatory parameters:

```
[ca]
default_ca = CA_default
[CA_default]
database                = index.txt
new_certs_dir           = issued
certificate             = root_cert.pem
private_key             = private/root_keypair.pem
```

The `database` parameter specifies the certificate index file, also known as the certificate database. It is a text file where each line has information about one certificate. The `new_certs_dir` parameter specifies a directory where newly issued certificates will be written. The `certificate` and `private_key` parameters specify the certificate and the corresponding private key of the current CA – in our case, the root CA.

5. Next, we will define some optional parameters. Our root CA will issue only intermediate CA certificates; hence, we can set the default issued certificate validity to 10 years:

```
default_days            = 3650
```

6. The following parameter sets the message digest algorithm to use for signing certificates issued by our root CA. There is a special `default` value for the `default_md` parameter. It means that the same message digest algorithm that was used in a signing certificate will be used in the signed certificate. For instance, if a signing certificate is signed using the SHA-256 hash function itself, the issued certificate will also use the SHA-256 hash function for its signature. In our case, this parameter will not make any difference because ED448 certificates always use the SHAKE256 message digest algorithm when signing other certificates. We will include this parameter here just to demonstrate the capabilities of the `openssl ca` subcommand:

```
default_md              = default
```

7. The next parameter instructs `openssl ca` to issue certificates with random serial numbers, as opposed to sequential serial numbers:

```
rand_serial             = yes
```

8. The subsequent parameter allows us to issue several certificates with the same Subject. It can be useful for replacing expired certificates:

```
unique_subject          = no
```

9. The next parameters control how certificate details are shown before signing. The OpenSSL documentation recommends setting these parameters – otherwise, an old broken display format will be used:

```
name_opt                = ca_default
cert_opt                = ca_default
```

10. The following parameter is mandatory and specifies the config section that defines the default issued certificate's Subject policy. We will review it more closely when we come to the section itself:

```
policy                  = policy_intermediate_cert
```

11. The next parameter specifies the default X509v3 extensions section:

```
x509_extensions         = v3_intermediate_cert
```

12. This next parameter instructs OpenSSL to copy the X509v3 extensions from the **Certificate Signing Request** (**CSR**) that are not added by the CA to the issued certificate. Selective copying allows a CSR to supply useful X509v3 extensions, such as *subjectAltName*, but forbids overriding important extensions set by the CA, such as *basicConstraints* or *keyUsage*:

```
copy_extensions         = copy
```

13. The next parameters are CRL-related. They set the default config section with CRL extensions, CRL number files, and the default validity of an issued CRL:

```
crl_extensions          = crl_extensions_root_ca
crlnumber               = crlnumber.txt
default_crl_days        = 30
```

14. The following parameters define the default Subject and X509v3 extensions for a CSR when the CSR is made with this config file. The only CSR made with this config file should be the CSR for the root CA certificate. Therefore, it is convenient to define the root CA certificate's Subject and X509v3 extensions in the config file, both for generating the CSR and, subsequently, the root CA certificate. Having certificate data in the config file is also helpful as a reminder about which CA certificate is used with it:

```
[req]
prompt                  = no
distinguished_name      = distinguished_name_root_cert
x509_extensions         = v3_root_cert
[distinguished_name_root_cert]
countryName             = NO
stateOrProvinceName     = Oslo
```

```
localityName            = Oslo
organizationName        = TLS Experts
commonName              = Root CA
```

15. The next config section defines the default issued certificate's Subject policy:

```
[policy_intermediate_cert]
countryName             = match
stateOrProvinceName     = match
localityName            = match
organizationName        = match
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = optional
```

As per the defined policy, we require the issued intermediate CA certificate's Subject to have the same geographical location and organization as the root CA certificate's Subject. It makes sense because we are running a mini-CA inside one organization.

16. The next config sections define the X509v3 extensions for the root CA certificate and the issued `intermediate` CA certificate. The required section can be selected using the `-extensions` command-line switch:

```
[v3_root_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:TRUE
keyUsage = critical, digitalSignature, cRLSign,
keyCertSign
crlDistributionPoints = URI:http://crl.tls-experts.no/
root_crl.der
[v3_intermediate_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:TRUE, pathlen:0
keyUsage = critical, digitalSignature, cRLSign,
keyCertSign
crlDistributionPoints = URI:http://crl.tls-experts.no/
root_crl.der
```

17. The last config section defines the CRL extensions that will be added to issued CRLs:

```
[crl_extensions_root_ca]
authorityKeyIdentifier = keyid:always, issuer
crlDistributionPoints = URI:http://crl.tls-experts.no/
root_crl.der
```

The full root CA config file can be found on GitHub as the `root.cnf` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter12/mini-ca/root/root.cnf`.

18. Our root CA config file is ready. Let's proceed to the next step, which is the generation of the root CA keypair:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/root_keypair.pem
```

19. The next step is generating the root CA CSR:

```
$ openssl req \
    -config root.cnf \
    -new \
    -key private/root_keypair.pem \
    -out root_csr.pem \
    -text
```

Note that we have not supplied the root certificate's Subject on the command line. The `openssl req` subcommand will figure out the Subject from the `root.cnf` config file, namely from the `req` and `distinguished_name_root_cert` sections.

20. After the CSR is ready, we can issue the root CA certificate:

```
$ openssl ca \
    -config root.cnf \
    -extensions v3_root_cert \
    -selfsign \
    -in root_csr.pem \
    -out root_cert.pem
Using configuration from root.cnf
Check that the request matches the signature
```

```
Signature ok
Certificate Details:
        Serial Number:
            ... (long hex number) ...
        Validity
            Not Before: Jul 18 18:54:11 2022 GMT
            Not After : Jul 15 18:54:11 2032 GMT
        Subject:
            countryName               = NO
            stateOrProvinceName       = Oslo
            localityName              = Oslo
            organizationName          = TLS Experts
            commonName                = Root CA
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                ... (long hex number) ...
            X509v3 Authority Key Identifier:
                ... (long hex number) ...
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Key Usage: critical
                Digital Signature, Certificate Sign, CRL
Sign
            X509v3 CRL Distribution Points:
                Full Name:
                  URI:http://crl.tls-experts.no/root_crl.
der
Certificate is to be certified until Jul 15 18:54:11 2032
GMT (3650 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Note that we have used the `openssl ca` subcommand instead of `openssl x509` in order to issue the certificate. Also, note that we have supplied most of the parameters in the config file instead of on the command line. We have not supplied the private key, duration, or X509v3 extensions on the command line – they all have been taken from the config file. We have used the `-selfsigned` switch to instruct `openssl ca` to issue a self-signed root CA certificate. We will not use that switch when issuing other certificates.

After issuing the certificate, `openssl ca` saves a copy of the issued certificate in the `issued` directory (specified by the `new_certs_dir` parameter in the configuration) and updates the certificate index file (specified by the `database` parameter in the configuration).

We have successfully generated the root CA certificate. In the next section, we will find out how to generate an intermediate CA certificate.

Generating an intermediate CA certificate

Generating an intermediate CA certificate will be similar to generating a root CA certificate. We will use another directory for our intermediate CA; let's call the directory `intermediate` and place it at the same level as the `root` directory inside the `mini-ca` directory:

1. Let's create the `intermediate` directory and initialize it with the needed files and subdirectory:

```
$ cd mini-ca
$ mkdir intermediate
$ cd intermediate
$ mkdir issued
$ echo -n >index.txt
$ echo 01 >crlnumber.txt
```

2. Now, we have to make a config file for our intermediate CA; we will name this file `intermediate.cnf`. The config file for the intermediate CA will be similar to that of the root CA. We will start the config file with the same parameters as in the root CA configuration:

```
[ca]
default_ca = CA_default
[CA_default]
database                = index.txt
new_certs_dir           = issued
certificate             = intermediate_cert.pem
private_key             = private/intermediate_keypair.
pem
default_days            = 365
```

```
default_md              = default
rand_serial             = yes
unique_subject          = no
name_opt                = ca_default
cert_opt                = ca_default
policy                  = policy_server_cert
x509_extensions         = v3_server_cert
copy_extensions         = copy
crl_extensions          = crl_extensions_intermediate_ca
crlnumber               = crlnumber.txt
default_crl_days        = 30
```

Note that the `default_days` parameter in this config file is lower than the same parameter in the root CA config file. It is because our intermediate CA will issue leaf certificates that usually have a shorter lifetime than root or intermediate CA certificates.

3. We will continue the config file with the parameters that will help to build the Subject of our intermediate CA certificate:

```
[req]
prompt                  = no
distinguished_name      = distinguished_name_
intermediate_cert
[distinguished_name_intermediate_cert]
countryName             = NO
stateOrProvinceName     = Oslo
localityName            = Oslo
organizationName        = TLS Experts
commonName              = Intermediate CA
```

4. Our intermediate CA is going to issue both server and client certificates. Hence, we need to define the Subject policies for those categories of certificates:

```
[policy_server_cert]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
```

```
emailAddress            = optional
[policy_client_cert]
countryName             = optional
stateOrProvinceName     = optional
localityName            = optional
organizationName        = optional
organizationalUnitName  = optional
commonName              = supplied
emailAddress            = supplied
```

As you can observe, the policies defined in the intermediate CA configuration are more relaxed than those in the root CA configuration. This is because we may potentially want to issue a certificate to an entity outside of our organization – for instance, to a customer or a partner – and because it is nice to have some variations in our demos.

5. The next sections in the intermediate CA configuration define X509v3 extensions that will be added to different kinds of issued certificates, such as server certificates, client certificates, and certificates for **Online Certificate Status Protocol** (**OCSP**) responders:

```
[v3_server_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:FALSE
nsCertType = server
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
crlDistributionPoints = URI:http://crl.tls-experts.no/
intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.
no/
[v3_client_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:FALSE
nsCertType = client, email
keyUsage = critical, nonRepudiation, digitalSignature,
keyEncipherment
extendedKeyUsage = clientAuth, emailProtection
crlDistributionPoints = URI:http://crl.tls-experts.no/
intermediate_crl.der
```

```
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.
no/
[v3_ocsp_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:FALSE
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning
crlDistributionPoints = URI:http://crl.tls-experts.no/
intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.
no/
```

6. The last section in the config file defines extensions that will be added to the issued CRLs:

```
[crl_extensions_intermediate_ca]
authorityKeyIdentifier = keyid:always, issuer
crlDistributionPoints = URI:http://crl.tls-experts.no/
intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.
no/
```

The full intermediate CA config file can be found on GitHub as the `intermediate.cnf` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter12/mini-ca/intermediate/intermediate.cnf`.

7. Our intermediate CA config file is ready. Now, let's generate the `intermediate` CA keypair and CSR:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/intermediate_keypair.pem
```

8. Next, generate the intermediate CA CSR:

```
$ openssl req \
    -config intermediate.cnf \
    -new \
    -key private/intermediate_keypair.pem \
    -out intermediate_csr.pem \
    -text
```

9. As with the root CA certificate, we have supplied the certificate Subject components from the config file instead of the command line.

10. The next step is to issue the intermediate CA certificate, but it must be issued by the root CA. Therefore, we will issue it from the `root` directory:

```
$ cd ../root/
$ openssl ca \
    -config root.cnf \
    -extensions v3_intermediate_cert \
    -in  ../intermediate/intermediate_csr.pem \
    -out ../intermediate/intermediate_cert.pem
Using configuration from root.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
        …
Certificate is to be certified until Jul 15 18:55:11 2032
GMT (3650 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Note that this time we have used X509v3 extensions from the `v3_intermediate_cert` section of the `root.cnf` file, unlike the previous time, when we used the `v3_root_cert` section for the root CA certificate.

We have now generated the intermediate CA certificate. In the next section, we will use it to issue a leaf certificate.

Generating a certificate for a web server

The next certificate that we will generate is a certificate for a web server. As for the previous certificates, we will make a separate directory and a config file for this certificate:

1. However, as we are going to issue a leaf certificate, we will not need to create CA-related files in its directory. We will just need to create the directory and change our current directory there:

```
$ cd mini-ca
$ mkdir server
$ cd server
```

The server certificate config file will be much shorter than the previously made CA certificate config files:

```
[req]
prompt                  = no
distinguished_name      = distinguished_name_server_cert
req_extensions          = v3_server_cert
[distinguished_name_server_cert]
countryName             = NO
stateOrProvinceName     = Oslo
localityName            = Oslo
organizationName        = TLS Experts
commonName              = internal.tls-experts.no
[v3_server_cert]
subjectAltName = DNS:mirror1.tls-experts.no, DNS:mirror2.
tls-experts.no
```

As you can observe, this time, we have defined an X509v3 extension, `subjectAltNames`, which is going to be added to the CSR. We have supplied several values for the extension and separated them using commas. There is also another method of supplying several values – by making a separate config section with each value on a separate line. We will review that method later when we create a client certificate. The mentioned extension is not present in the intermediate CA configuration, and the intermediate CA configuration contains the `copy_extensions = copy` option. Hence, the X509v3 extension from the CSR will be copied to the resulting certificate together with other extensions that are defined in the intermediate CA config file. Of course, when issuing the certificate, we will be able to review the certificate details and see all the included X509v3 extensions.

2. The next step is to generate a keypair and a CSR for our server certificate:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/server_keypair.pem
$ openssl req \
    -config server.cnf \
    -new \
    -key private/server_keypair.pem \
    -out server_csr.pem \
    -text
```

Note that we don't have to supply the `-reqexts v3_server_cert` option on the command line, because `v3_server_cert` is the default extensions section, specified by the `req_extensions = v3_server_cert` option in the config file.

3. Now, it's time to generate our server certificate. As it will be issued by the intermediate CA, we will run the `openssl ca` subcommand from the `intermediate` directory:

```
$ cd ../intermediate/
$ openssl ca \
    -config intermediate.cnf \
    -in  ../server/server_csr.pem \
    -out ../server/server_cert.pem
Using configuration from intermediate.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
        …
Certificate is to be certified until Jul 19 15:36:44 2023
GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

Look how easy the certificate-issuing command was! Just three arguments. All the other parameters were taken from the configuration, including the default policy and the X509v3 extensions. Having configured everything in the config file, we can use such a simple command and issue server certificates with ease.

Issuing a client certificate is a little bit different, but also not very difficult. We will learn how to do so in the next section.

Generating a certificate for a web and email client

Generating a client certificate is similar to generating a server certificate:

1. As usual, we will make a separate directory for this certificate and change our current directory there:

```
$ cd mini-ca
$ mkdir client
$ cd client
```

2. We will use the following configuration saved in the `client.cnf` file for generating the client certificate:

```
[req]
prompt                  = no
distinguished_name      = distinguished_name_client_cert
req_extensions          = v3_client_cert
[distinguished_name_client_cert]
countryName             = NO
stateOrProvinceName     = Oslo
localityName            = Oslo
organizationName        = TLS Experts
commonName              = Thor Odinson
emailAddress            = thor@tls-experts.no
[v3_client_cert]
subjectAltName = @subject_alt_names
[subject_alt_names]
email.1 = postmaster@tls-experts.no
email.2 = hostmaster@tls-experts.no
```

   As last time, we use the `subjectAltName` extension with multiple values, but now we supply several values using a separate config section.

3. Let's generate the certificate keypair and the CSR:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/client_keypair.pem
$ openssl req \
    -config client.cnf \
    -new \
    -key private/client_keypair.pem \
    -out client_csr.pem \
    -text
```

4. The next step is to issue the client certificate:

```
$ cd ../intermediate/
$ openssl ca \
    -config intermediate.cnf \
    -policy policy_client_cert \
    -extensions v3_client_cert \
    -in  ../client/client_csr.pem \
    -out ../client/client_cert.pem
Using configuration from intermediate.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
        …
Certificate is to be certified until Jul 19 15:37:44 2023
GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

For the client certificate, the issuing command is slightly longer than for the server certificate because we have to supply the `-policy` and `-extensions` switches. The default policy and extension sections are reserved for server certificates because they are issued more often than client certificates. However, if it is not the case in your organization, you can very well use client policy and extension sections as defaults.

5. One more thing that can be useful when making a client certificate is packaging the certificate, its private key, and its verification chain into a **Public Key Cryptography Standards #12** (**PKCS #12**) container:

```
$ cd ../client/
$ cat \
    ../intermediate/intermediate_cert.pem \
    ../root/root_cert.pem \
    >certfile.pem
$ openssl pkcs12 \
    -export \
    -inkey private/client_keypair.pem \
    -in client_cert.pem \
    -certfile certfile.pem \
    -passout 'pass:SuperPa$$w0rd' \
    -out client_cert.p12
```

As we see, issuing certificates with `openssl ca` is rather easy. But how do we revoke certificates and generate CRLs? Let's find out in the next section.

Revoking certificates and generating CRLs

Before revoking a certificate, we must issue it first. Let's issue a sample certificate, similar to how we did so before:

1. As usual, the first step is to make a directory for the certificate files:

```
$ cd mini-ca
$ mkdir server2
$ cd server2
```

2. Next, make the following certificate configuration and save it to the `server2.cnf` file:

```
[req]
prompt                  = no
distinguished_name      = distinguished_name_server_cert
[distinguished_name_server_cert]
countryName             = NO
stateOrProvinceName     = Oslo
localityName            = Oslo
```

```
organizationName        = TLS Experts
commonName              = server2.tls-experts.no
```

3. Next, make the keypair, the CSR, and issue the certificate:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/server2_keypair.pem
$ openssl req \
    -config server2.cnf \
    -new \
    -key private/server2_keypair.pem \
    -out server2_csr.pem \
    -text
$ cd ../intermediate/
$ openssl ca \
    -config intermediate.cnf \
    -in  ../server2/server2_csr.pem \
    -out ../server2/server2_cert.pem
Using configuration from intermediate.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
        …
Certificate is to be certified until Jul 19 15:38:44 2023
GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

We have just generated a new certificate. We are still in the `intermediate` directory. Let's generate a CRL when the `server2` certificate is not revoked yet first.

4. This is how we generate a CRL for our intermediate CA:

```
$ openssl ca \
    -config intermediate.cnf \
    -gencrl \
    -out intermediate_crl.pem
```

5. Let's view the generated CRL as text:

```
$ openssl crl \
    -in intermediate_crl.pem \
    -noout \
    -text
Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: ED448
        Issuer: C = NO, ST = Oslo, L = Oslo, O = TLS
Experts, CN = Intermediate CA
        Last Update: Jul 19 16:56:47 2022 GMT
        Next Update: Aug 18 16:56:47 2022 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                ... (hex values) …
            X509v3 CRL Distribution Points:
                Full Name:
                  URI:http://crl.tls-experts.no/
intermediate_crl.der
            Authority Information Access:
                OCSP - URI:http://ocsp.tls-experts.no/
            X509v3 CRL Number:
                1
No Revoked Certificates.
    Signature Algorithm: ED448
    Signature Value:
        ... (hex values) ...
```

We have just generated our first CRL with an `X509v3 CRL Number` value of `1`. This number has been taken from the `crlnumber.txt` file, which is configured by the `crlnumber = crlnumber.txt` line in the intermediate CA `config` file. That file contains the next CRL number in `hex`. After a CRL is issued, `openssl ca` increments the number in that file by one.

6. As we can see, the CRL states that no certificates are revoked. Let's revoke the `server2` certificate and regenerate the CRL.

   That is how we revoke a certificate:

```
$ openssl ca \
    -config intermediate.cnf \
    -revoke ../server2/server2_cert.pem \
    -crl_reason keyCompromise
Using configuration from intermediate.cnf
Revoking Certificate
1651F3139172DEE541B914DFCB371D8E11BA209F.
Data Base Updated
```

   The revocation information is saved in the certificate index file, `index.txt`. The certificate status in that file is changed from `V` (valid) to `R` (revoked).

7. After the certificate is revoked, we have to regenerate the CRL:

```
$ openssl ca \
    -config intermediate.cnf \
    -gencrl \
    -out intermediate_crl.pem
```

8. Let's inspect the updated CRL:

```
$ openssl crl \
    -in intermediate_crl.pem \
    -noout \
    -text
Certificate Revocation List (CRL):
        Version 2 (0x1)
        Signature Algorithm: ED448
        Issuer: C = NO, ST = Oslo, L = Oslo, O = TLS
Experts, CN = Intermediate CA
        Last Update: Jul 19 17:43:49 2022 GMT
        Next Update: Aug 18 17:43:49 2022 GMT
        CRL extensions:
```

```
            X509v3 Authority Key Identifier:
                ... (hex values) …
            X509v3 CRL Distribution Points:
                Full Name:
                  URI:http://crl.tls-experts.no/
intermediate_crl.der
            Authority Information Access:
                OCSP - URI:http://ocsp.tls-experts.no/
            X509v3 CRL Number:
                2
Revoked Certificates:
    Serial Number:
1651F3139172DEE541B914DFCB371D8E11BA209F
        Revocation Date: Jul 19 17:41:39 2022 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    Signature Algorithm: ED448
    Signature Value:
        ... (hex values) ...
```

As we can observe, now the `X509v3 CRL Number` value is `2` and the CRL contains one revoked certificate. We can conclude that the revocation of certificates and the generation of CRLs work as expected.

9. Another thing that is useful to do is to convert the issued CRL from the **Privacy-Enhanced Mail** (**PEM**) format to the **Distinguished Encoding Rules** (**DER**) format because CRL distribution points usually serve CRLs in the `DER` format:

```
$ openssl crl \
    -in intermediate_crl.pem \
    -out intermediate_crl.der \
    -outform DER
```

We have now learned how to revoke certificates and generate CRLs. As we know, serving CRLs is not the only method of serving information about certificate revocation. Another method is OCSP. Let's learn a little bit more about OCSP in the next section.

Providing certificate revocation status via OCSP

To serve OCSP responses, we have to sign them. An OCSP response for a certificate can be signed by its issuer certificate. The same issuer can also issue another certificate for signing OCSP requests. That certificate must have `OCSPSigning` included in the X509v3 `extendedKeyUsage` extension.

When we created the intermediate CA config file, we included the following X509v3 extensions section:

```
[v3_ocsp_cert]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always, issuer
basicConstraints = critical, CA:FALSE
keyUsage = critical, digitalSignature
extendedKeyUsage = critical, OCSPSigning
crlDistributionPoints = URI:http://crl.tls-experts.no/
intermediate_crl.der
authorityInfoAccess = OCSP;URI:http://ocsp.tls-experts.no/
```

That X509v3 extensions section will help us to generate a certificate for an OCSP responder. Let's make this certificate:

1. As usual, we will start by preparing the directory:

```
$ cd mini-ca
$ mkdir ocsp
$ cd ocsp
```

2. Then, we will make a configuration for the OCSP responder certificate and save it to the `ocsp.cnf` file:

```
[req]
prompt                  = no
distinguished_name      = distinguished_name_ocsp_cert
[distinguished_name_ocsp_cert]
countryName             = NO
stateOrProvinceName     = Oslo
localityName            = Oslo
organizationName        = TLS Experts
commonName              = OCSP Responder
```

3. Using the saved configuration, we will create the OCSP responder certificate:

```
$ mkdir private
$ chmod 0700 private
$ openssl genpkey \
    -algorithm ED448 \
    -out private/ocsp_keypair.pem
$ openssl req \
    -config ocsp.cnf \
    -new \
    -key private/ocsp_keypair.pem \
    -out ocsp_csr.pem \
    -text
$ cd ../intermediate/
$ openssl ca \
    -config intermediate.cnf \
    -in  ../ocsp/ocsp_csr.pem \
    -out ../ocsp/ocsp_cert.pem
Using configuration from intermediate.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
        …
Certificate is to be certified until Jul 19 17:45:44 2023
GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

The generated certificate can be used by an OCSP responder. Let us test it. The `openssl` utility includes the `openssl ocsp` subcommand, which can act as a simple OCSP server. We can use such an OCSP server for our testing. That OCSP server takes certificate validity data right from the certificate index file, `index.txt`. Very practical for testing, if you ask me.

4. You can read the `openssl ocsp` documentation on its `man` page:

```
$ man openssl-ocsp
```

5. We will start our test OCSP server from the `ocsp` directory:

```
$ cd ../ocsp/
$ openssl ocsp \
    -port 4480 \
    -index ../intermediate/index.txt \
    -CA ../intermediate/intermediate_cert.pem \
    -rkey private/ocsp_keypair.pem \
    -rsigner ocsp_cert.pem
ACCEPT 0.0.0.0:4480 PID=124271
ocsp: waiting for OCSP client connections...
```

   We have provided our OCSP responder certificate and its private key to `openssl ocsp` for signing the OCSP responses.

6. But the `openssl ocsp` subcommand is so nice that it can act not only as an OCSP server but also as an OCSP client. Let's use it in client mode and check the validity of the `server` certificate that we recently issued. To do so, we need to open another terminal window, change to the `mini-ca/ocsp` directory, and run the following command:

```
openssl ocsp \
    -url http://localhost:4480 \
    -sha256 \
    -CAfile ../root/root_cert.pem \
    -issuer ../intermediate/intermediate_cert.pem \
    -cert ../server/server_cert.pem
Response verify OK
../server/server_cert.pem: good
        This Update: Jul 19 19:59:06 2022 GMT
```

   As we can observe, the OCSP server has confirmed the validity of the `server` certificate.

7. Now, let's check the status of the `server2` certificate that we have recently issued and revoked:

```
$ openssl ocsp \
    -url http://localhost:4480 \
    -sha256 \
    -CAfile ../root/root_cert.pem \
    -issuer ../intermediate/intermediate_cert.pem \
    -cert ../server2/server2_cert.pem
Response verify OK
```

```
../server2/server2_cert.pem: revoked
        This Update: Jul 19 20:03:56 2022 GMT
        Reason: keyCompromise
        Revocation Time: Jul 19 17:41:39 2022 GMT
```

As we can see, the OCSP responder has reported that the `server2` certificate is revoked, just as we expected. We can conclude that our OCSP setup works correctly.

We have now learned a lot about running a mini-CA. We have used many config files and commands. Those config files and commands, saved as Shell scripts, can be found on GitHub, for trying out or for future reference: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter12`.

This brings us to the end of this chapter. Now, we'll summarize everything we have learned here.

Summary

In this chapter, we learned how to run a mini-CA. First, we learned about why a mini-CA can be useful within an organization. We also learned about the `openssl ca` subcommand and how to make configuration files for it. Then, we learned how to issue certificates using `openssl ca`. After that, we learned how to revoke certificates and issue CRLs. We finished the chapter by learning how to issue a certificate for an OCSP responder and how to provide certificate revocation status via OCSP using the `openssl ocsp` subcommand. This knowledge can help you to set up and run a mini-CA, gaining control over PKI in your organization.

This was the last chapter of the book. I hope that you have enjoyed both the chapter and the book, and have learned something new and useful. I also hope that the knowledge gained will help you to understand cryptographic and network security technologies better, develop more secure applications, and advance your career.